5-31-2018

IPv6 Security Issues in Linux and FreeBSD Kernels: A 20-year Retrospective

Jack R. Cardwell
Dartmouth College

Dartmouth College Computer Science
Technical Report TR2018-843

IPv6 Security Issues in Linux and FreeBSD Kernels: A 20-Year Retrospective

Jack Cardwell

Advisor:
Sergey Bratus

May 31, 2018

Abstract

Although IPv6 was introduced in 1998, its adoption didn't begin to take off until 2012. Furthermore, its vulnerabilities haven't received as much attention as those of IPv4. As such, there is potential to exploit these vulnerabilities. With the amount of IPv6 traffic rapidly increasing, these exploits present real-world consequences. This paper aims to re-evaluate the security of IPv6 stack implementations in FreeBSD and Linux kernels, specifically FreeBSD 11.1 and Ubuntu Linux 4.13. It contributes to the literature in three ways. We first reproduce ten vulnerabilities from existing research to determine whether known bugs have been patched. Then, we examine two, new vulnerabilities in IPv6 extension headers and options. Not only does this paper demonstrate the vulnerabilities in the kernels' implementations, but it also aims to show where these parser differentials likely originate in the kernel's source code. Our hope is that the fuzzing cases from this paper can be built into an automatic fuzzing framework that will facilitate the discovery of new vulnerabilities and ensure the security of this protocol moving forward.

1 Introduction

Internet Protocol Version 6 was developed and introduced by the Internet Engineering Task Force (IETF) in 1998 to replace the existing protocol, version 4 [1]. Primarily, it responded to the shrinking IPv4 address space that was still available, expanding the address size from 32 bits to 128 bits. RFC 2460 also provides three other rationales for the development of IPv6: the simplification of IP headers, the improvement of support for options and extensions, and finally the capability to label packet flows [1]. The IETF writes that early adopters of IPv6 address space may have been influenced by the relatively cheaper cost of a gradual IPv6 deployment, when compared to the enormous costs of a rapid deployment when IPv4 space eventually becomes depleted. The same report states that an IPv4 address was near its max projected cost in 2017 and that the cost of maintaining the stability of an IPv4 address was also rising, given the relative complexity of IPv4 headers [2].

IPv6 headers represent a dramatic shift away from those of their predecessor, shifting many of the IPv4 header fields into extension headers and options. This likely happened due to errors in handling IPv4 options, some of which were officially decremented in RFC6814 [3]. Furthermore, IPv6 did away with broadcast addresses, as in IPv4. Instead, it switched to a system of *multicast*, *unicast*, and *anycast* addresses. Along with this change, IPv6 also switched from *Address Resolution Protocol* (ARP) to *Neighbor Discovery Protocol* (NDP) to accomplish address resolution. This change is especially significant because it switched responsibility of network configuration to ICMPv6. In order to facilitate the transition between versions, some providers have implemented dual IPv4/IPv6 stacks, which was deemed best practice by RFC 6540 [5]. Additionally, tunneling IPv6 traffic over IPv4 connections has become prevalent [4]. While these have eased the transition from IPv4 to IPv6, this traffic will eventually switch to IPv6, further increasing the need for its stability.

1.1 Security Concerns

Since IPv6 was introduced in 1998, the IPv6 header has been researched and some vulnerabilities have been documented. Figure 1 shows a graph of all documented vulnerabilities in the Common Vulnerabilities and Exposures

(CVE) database, categorized by year.[1] These vulnerabilities were found by running a keyword search for *ipv6*. There are two reasons why the number of vulnerabilities may fluctuate with time. First, researchers may become better at discovering new attacks. Second, the kernel source code may be evolving, opening up new areas of the code to attack. The bars in Figure 1 represent the lines of code in the FreeBSD *netinet6* and the Linux *ipv6* directories. The graph shows that there is no clear relationship between the size of the kernel code and the number of vulnerabilities found in each year.

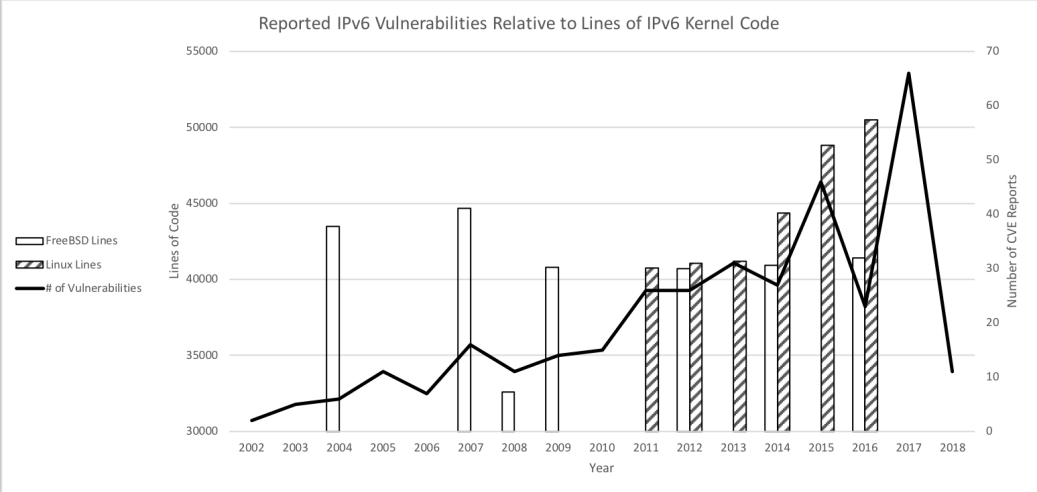

Figure 1: Reported IPv6 Vulnerabilities Relative to Lines of IPv6 Kernel Code

This section aims to identify the known vulnerabilities and summarize the existing research that has been done on both the Linux and FreeBSD kernels.

Router and Neighbor Discovery Attacks

In order to configure a new appliance with an IPv6 address, the machine must first obtain a unique address in the network. This step must be taken regardless if the device is manually configured, stateless auto-configured, or

[1] http://cve.mitre.org

stateful auto-configured. *Neighbor Discovery Protocol* (NDP) is used to accomplish this. Specifically, when a device first joins a network, it sends out a Neighbor Solicitation message with the requested IPv6 address to all other nodes on the same network. This exploit was documented in CVE-2008-2476 and Lecigne et al. [7] [6]. The authors demonstrate that this protocol is easily exploitable. Simply by sniffing on the network and responding to each Neighbor Solicitation message with a spoofed Neighbor Advertisement, no new hosts can gain access to the network. Lecigne and Neville-Neil demonstrate this vulnerability in FreeBSD in their 2006 paper, but the same logic should hold in Linux [6].

Neighbor Advertisements are also susceptible to a similar attack that overwrites the cache that maps each neighbor to its link-layer address. An attacker can spoof a Neighbor Advertisement message with the override flag set in the packet. So long as the packet includes a valid link-layer address, this packet can overwrite the victim's cache, such that all packets sent to the spoofed source address are directed to the attacker's link-layer address, instead of the intended destination's link-layer address. This attack was previously carried out on Windows machines, as documented in CVE-2007-1532 [8].

Because of the host's ability to independently configure itself, Router Advertisements fall victim to the same sort of logic. An attacker can sit on a network, sniffing for valid Router Solicitation packets from other hosts. When the attacker detects a Router Solicitation packet, it can respond with its own address, such that it sits in the middle of all packets from the victim. Furthermore, the attacker could cause a denial of service if it inserts a non-existent router address. However, without any recirculation of these malformed packets, these attacks are only effective for a certain period of time, until the victim's routing table times out sends a new Router Solicitation message.

ICMPv6 Attacks in IPv6

Internet Control Message Protocol Version 6 (ICMPv6) functions very similarly to its IPv4 counterpart, often used to send error messages between hosts to configure a connection. Lecigne et al. 2006 detail some of the ways to manipulate these messages in order to exploit the IP stream [6]. These attacks are effective because RFC1122 states that all TCP connections must act on ICMP error messages passed up from the IP layer [9].

The first way that ICMPv6 can be used maliciously is to shrink the Maximum Transfer Unit (MTU) of a connection in order to slow the traffic between two hosts. This is easily achieved by sending an ICMPv6 Too Big Message from the attacker to the victim. This attack was documented by Gont in his internet draft [10]. Per RFC 2460, all MTUs must be a size of 1280 octets or higher within an ICMPv6 Too Big packet [1]. Otherwise, the packet should be discarded. These ICMPv6 Too Big packets are used during the Path MTU discovery process to determine how large a packet can be when sent to a node, which is well documented in RFC 1981 [11]. Previous research has sought to verify whether kernels properly verify MTU and whether the results are as expected. If the victim did not verify the MTU, the attacker could slow traffic and potentially cause a DoS. According to CVE-2008-3530, both FreeBSD and NetBSD were previously vulnerable to this type of attack [12].

In addition to slowing down a connection between two hosts, ICMPv6 messages can be used in such a way to terminate the same connection. Specifically, an attacker can insert an ICMPv6 Destination Unreachable message with a number of different options. There are many reasons why a spoofed ICMPv6 Destination Unreachable message may stand out as unusual to the end node, but not all kernels check these cases. Gont discusses these cases in his internet draft "ICMP Attacks Against TCP" [10]. For instance, it wouldn't make much sense if a TCP connection is in the *CONNECTED* state and the victim receives a Destination Unreachable message, with a Protocol Unreachable option. Lecigne et. al 2006 verify that FreeBSD is not vulnerable to this attack, but it is possible that Ubuntu Linux 4.13 or versions of FreeBSD, past 11.1, are susceptible [6].

Spoofed Router Advertisements are one way to sit in the middle of a connection, but ICMPv6 offers another way to achieve the same outcome. An attacker can start by sniffing for any traffic on an interface. When it sees some traffic between two connections, it can send an ICMPv6 Redirect message, informing the victim that it needs to update its routing table with the address specified in the message. If the attacker uses its own link-layer address and the victim trusts the message, then the attacker can view all traffic that is sent from the victim. This vulnerability, along with many of these systemic vulnerabilities, may not be easily patched, short of ignoring ICMPv6 Redirect messages. RFC 4301 states that each kernel's implementation must be configurable, such that some of these messages can be ignored [13]. However, this configuration could degrade services, such as prohibiting

the discovery of the best route for a message.

Other Attacks

There are a few other parser attacks that have been explored, which do not quite fit into the above categories. The first attack is perhaps the most well-known hack on a network stack: the Ping of Death. This attack fragments an ICMPv6 Echo Request that, when chained together, is larger than the maximum size of an IP packet, 65535 bytes. Because this packet is fragmented, each individual packet in the sequence would appear legitimate to the kernel. If the victim's stack does not properly check fragment lengths before reassembly, this attack could overwrite the receive buffer or cause the kernel itself to crash.

Another attack that has been widely discussed, the Rose Attack, revolves around sending fragments of a packet in an attempt to overwhelm the victim's receive buffer. To accomplish this attack in IPv6, one must make use of the Fragment Extension header. When a valid message is fragmented and sent, the receiving stack saves the packets into a buffer and waits until all pieces of the message are present before reassembling them. In doing so, the stack may allocate memory for fragments that it has not yet received, but expects to see, based on the sequence numbers. The Rose Attack consists of sending a number of fragments with different sequence numbers, such that there are many gaps left in the packet that might be allocated for. Theoretically when enough of these packets are sent, the victim will reject all other traffic because its receive buffer is full [6]. Therefore, by simply controlling which packets are sent, an attacker could cause a DoS. Lecigne et. al show that FreeBSD is not vulnerable to this attack, but the results could differ in newer versions of the kernel or in Linux [6]. This paper replicates these tests in order to determine whether the systemic vulnerabilities have been patched.

2 Contributions of this Paper

This paper contributes to the current literature in three primary ways. First, this paper will re-evaluate the vulnerabilities that were discussed in the previous section. By testing known vulnerabilities, this paper shows whether bugs discovered roughly twelve years ago in Lecigne et. al 2006 have been patched in FreeBSD 11.1 and Ubuntu Linux 4.13. The results could also

change because networks are now faster, which makes it more difficult to insert packets before the true network responses reach their destinations.

Next, this paper furthers the study in this field by considering a number of new parser differentials. Specifically, this paper presents two malformed packets that are handled differently by the FreeBSD and Linux kernels. By presenting these parser differentials, this paper offers a tool to profile a victim's network stack, based on a number of different edge cases. These parser differentials could also be used to evade network intrusion detection systems (NIDs). Through these two methods, this paper offers a glimpse into the state of IPv6 implementation in two popular network stacks. The findings of this paper are summarized in Table 1. Rows 1-10 of Table 1 are attacks that are replicated from existing literature, while rows 11 and 12 are new tests, unique to this thesis.

This paper finally contributes to the literature by providing rationale for an automatic IP stack fuzzer. We draw the distinction that an automatic fuzzer is able to interpret stack's responses and adapt new packets, independent of any human interaction, while an automated fuzzer still requires human intervention. Many fuzzers exist, such as American Fuzzy Lop (AFL) and The Hacker's Choice toolkit, but these programs often rely on a default set of rules or are not completely automatic.[2] [3] Other implementations rely on randomly shifting bits in a IP header. In this sense, the existing programs are unintelligent and could waste time testing senseless cases. This paper suggests that there are intelligent ways to fuzz a network stack that could uncover new vulnerabilities and enhance the security of the network stack.

3 Packages and Tools

This project made use of many different packages and tools in order to test the two different network stacks. Most fundamentally, this project depends on running the two different kernels. Oracle's VirtualBox was used as an environment to test the kernels' responses to these malformed packets in an internal network. One drawback to this approach is that the internal virtual network lacks the middlemen such as firewalls that could impact the delivery

[2] http://lcamtuf.coredump.cx/afl/

[3] https://github.com/vanhauser-thc/thc-ipv6

of a packet to the end node. However, since the goal of this paper is to investigate the behavior at the end node, this downside is not significant.

Three virtual machine images were created for the VirtualBox environment. The first machine, the source of all malformed packets in this setup, was running Ubuntu Linux 4.8.0. The two victims' machines, whose stacks were being fuzzed, ran FreeBSD 11.1 and Ubuntu Linux 4.13. All of these virtual images were created from source code, such that the network stack could eventually be traced with the same source code.

Packet creation was facilitated through the usage of Scapy, a Python-built module that extended its library to handle IPv6 packets in 2012. [4] This package saved countless lines of code and allowed the project to focus on the packets' contents, instead of the packets' raw bytes. The Scapy library was run over Python 2.7.12 on the Linux host machine. *Tcpdump* and *Wireshark* were also installed on this machine in order to track responses to the packets being sent from the fuzzer. Finally, all fuzzing scripts are publicly available on GitHub. [5]

On the victim's side of the connection, there were a few software dependencies required in order to track the kernel's response to the packet. These packages allow one to profile the kernel stack, visualizing how far into the stack a malformed packet travels, before it is discarded, if at all. For FreeBSD, *dtrace* was used. *Dtrace* is a dynamic tracer developed by a Brendan Gregg and a few software developers at Sun Microsystems. [6] On Linux, *kprobe* was used, in place of *dtrace*. *Kprobe* allows the user to insert tracepoints within the kernel code. [7] For instance, one could trace the kernel function *ip6_input()* and follow the children calls, using this library.

4 Methodology

As mentioned in the previous section, this experiment depended on three different virtual machines running on an internal network. To begin this experiment, it was necessary to record a baseline response from both the FreeBSD 11.1 and the Ubuntu Linux 4.13 kernels. The purpose of this was twofold: first, it verified connectivity between hosts on the network; second,

[4] https://scapy.net
[5] https://github.com/jcardwell/ExploitScripts
[6] http://dtrace.org/blogs/
[7] https://www.kernel.org/doc/Documentation/kprobes.txt

it showed the "normal" path of a packet through each kernel's source code. The baseline packet in this setup was an IPv6 packet with an ICMPv6 Echo Request that was sent to both the Linux and the FreeBSD kernels. Ping requests were well-suited for this purpose because all correctly formed, outbound packets should receive a response, provided there is no loss in the network. Because this experiment ran on an internal network with no packet loss, any unseen response can be attributed to a victim's failure to process the inbound packet.

Before sending any packets, the tracing utilities were configured to track kernel calls from *ip6_input()*, as well as any sub-calls that originate from children of this function. Output from both *kprobe* and *dtrace* was re-routed to a file for further investigation after the packets had been received. These tracers were enabled just before sending the packet of interest and disabled shortly after this packet had been sent, such that no other traffic would be collected for analysis. Once the tracing was set up, Wireshark was run on the attacker's machine. This verified that the correct packet was sent and was an easy way to monitor whether the victim responded to the packet. Once the baseline test was conducted with a correctly formed packet, a number of different edge cases were tested. All of the scripts for the following attacks were inspired by tools from The Hacker's Choice.[8]

4.1 Tests of Systemic Vulnerabilities

All ten attacks described in this section are replicated from existing literature to see if the responses to these attacks have changed since the release of Lecigne et. al in 2006.

Neighbor Solicitation

This attack sniffs the network for any Neighbor Solicitation messages that would be sent when a new host is trying to bring up an interface with a certain address. Whenever a machine attempts to use a certain address, it must verify that this address isn't yet taken on the network. When the program encounters a Neighbor Solicitation message, it forges and sends a Neighbor Advertisement message with the address in question set as the source. While this vulnerability does not involve any malformed packets, it

[8]https://github.com/vanhauser-thc/thc-ipv6

checks whether or not the victim can detect and handle spoofed messages. If the victim processes and accepts the Neighbor Advertisement, this exploit could cause a Denial of Service.

Neighbor Discovery Protocol

This attack sends a stream of Neighbor Advertisements to the network, with the override flag set and an optional extension header—ICMPv6 Neighbor Discovery Option Destination Link Local Address. The hope of this attack is that these override flags will cause the victim to overwrite its neighbor cache with the attacker's link-local address, in place of the destination's actual link-local address. Thus, all traffic for another target from the victim would be improperly sent to the attacker, instead of the intended target.

Router Advertisement

When a machine sends a packet to a destination, it looks to see which routers are available to transport the packet. Some of this information is cached in a routing table that is updated periodically with router advertisements. This process was designed in RFC 8106 [14]. When no routing information is available to a host or if a route times out, a node might send out a Router Solicitation message, asking for a way to send its packets. This attack takes advantage of these messages by sniffing for them and responding to them with a spoofed Router Advertisement response. This was a similar attack to that described in CVE-2011-2393 [15]. In these packets, the attack sets the override bit and sets the router lifetime to the maximum permitted value in the Scapy library (9000 seconds). These measures guarantee that the exploit has the maximum effect, if the packet penetrates the victim's stack.

MTU Poisoning

Under IPv6, routers and other middleware are not permitted to fragment a packet. Rather, the size of a packet must be determined at either end of a connection. This size is referred to as the path Maximum Transmission Unit (MTU). When a connection first begins, a host will send a packet of a certain size to the receiver. If the receiver can handle the length of the packet, no error occurs and the packet is processed. However, if the receiver can't handle a message of a certain size, then the receiver sends back an ICMPv6 Packet Too Big message. If a host receives this message, it is forced

fragment the packet "into a series of fragments each with a size less than or equal to the path MTU" [16]. According to the current standard, all nodes on a network must support a minimum MTU of 1280 bytes [1]. This attack exploits this connection by sniffing for traffic on a network and responding to any connection with a Packet Too Big message that contains a very small MTU. CVE-2015-8215 describes this attack [17]. In this experiment, an MTU of 800 bytes was used. If the victim were to process this packet and accept the MTU, it would slow the connection because more packets must be sent and processed. Therefore, this attack carries the potential to cause a Denial of Service. By the proposed standard, "if a node receives an ICMPv6 Packet Too Big message reporting a next-hop MTU that is less than the IPv6 minimum length MTU, it must discard it" [16].

Destination Unreachable

This attack is another way to potentially cause a Denial of Service in a more direct manner than the MTU poisoning method. When transporting a packet on the network, if a router detects no path to the destination, that machine will send an ICMPv6 Destination Unreachable method back to the source of the packet [18]. This test case sniffs the network for any traffic coming through a machine and responds to all traffic with an ICMPv6 Destination Unreachable message. To set up this test, the Linux and FreeBSD victims began to ping a different node on the internal network before the script was run. If the kernel were susceptible to this attack, the pings would return an error which would be visible in the terminal of the victim's machine. This test was influenced by the vulnerability reported in CVE-2005-0068 [19].

Redirection

FreeBSD and Linux kernels could be susceptible to a man in the middle attack that utilizes ICMPv6 Neighbor Discovery Redirect messages. This attack listens to all traffic on a network and responds to every packet with a redirect message that attempts to redirect messages to a user-specified address. If the attacker uses his own address, then he could potentially listen to all traffic in a connection. A kernel may not comprehensively check these redirect messages, such that it unnecessarily redirects messages to an attacker, in place of a valid router. This attack has been successful in Cisco systems according to CVE-2014-2144, but it has yet to be reported in FreeBSD or

Linux [20].

Ping of Death

This attack creates a packet of random data that, when aligned as one packet, is above the legal transport size for IPv6 (65535 octets). Then, it fragments this data and attempts to send these packets to the victim. If the FreeBSD or Linux stack mishandles the length of these fragments and attempts to reassemble them, there is potential for the stack to crash or for the buffer to overflow.

New Rose Attack

Just like the previous test, this attack attempts to expose kernels that mishandle fragments. Within each IPv6 Extension Header Fragment, there is a field for offset, which states where in the sequence of the whole packet the fragment exists. In correctly formed packets, an offset should equal the previous offset plus the length of the previous packet. In this way, the receiving stack can reassemble the packet. This attack sends packets with random offsets that do not line up, such that the receiver's buffer could fill up waiting for unseen packets to show up. Therefore, the receiver's stack could crash or could stop listening to incoming traffic.

Land Attack

This attack targets the transport layer, specifically TCP, that is embedded in the IPv6 connection. It creates a number of SYN packets that set both the source and the destination address as the victim's address. It also specifies two random ports to try on the victim's machine: one is set as the source part; the other, as the destination port. Therefore, if this packet were to be processed, the victim would send SYN and SYN-ACK packets between two of its own ports. In order to avoid classification as a duplicate packet, the sequence number of the SYN packet was randomized. This attack could cause a denial of service, if the victim treats the packets as valid and thus can't take on anymore TCP connections.

Source Smurfing

This attack represents an attempt to cause a DoS on the receiver, by overwhelming it with traffic. This attack makes use of a simple ICMPv6 Echo Request Packet that is sent to the victim. The source address is set as a multicast address for all nodes on a network. In a large network with thousands of nodes, the large number of responses may be enough to overwhelm the victim, if not just to slow down its response rate. Numerous RFCs have specified that Echo Request packets with a multicast source should be discarded, but some stacks may have disregarded this warning [1]. This attack could be repeatedly carried out to generate more traffic on the network, but this script only sends out one ICMPv6 Echo Request message.

4.2 Tests of Parser Vulnerabilities

The previous tests focus on systemic vulnerabilities that are not caused by malformed packets, but instead by logical errors in how packets are processed. This section discusses malformed packets that manage to escape detection and are processed by the victim. While the intents of the previous exploits are often clear—DoS or Man-in-the-Middle—this section discusses how the packets are mishandled, but doesn't focus on how they could be used to attack a kernel. In some of these cases, a network intrusion device (NID) would disregard a packet as nonsense, believing that the receiver wouldn't accept it. However, as this section shows, the receiver's response could differ from what the NID anticipates allowing for undetected packet insertion in a connection.

These vulnerabilities were uncovered by examining the source code of each kernel. After conducting a code review, two areas seemed vulnerable: extension header processing and options processing within these extension headers. When IPv6 first became the standard, extension headers were defined by a looser set of guidelines. RFC 7045 was published as an update for RFC 2460 in order to provide more standards on how these extension headers should be implemented and processed [21] [1]. Perhaps due to this delay in standards, the FreeBSD and Linux kernels were slow to evolve and error handling of extension headers and options became vulnerable. Two different methodologies were used to find these parser vulnerabilities. First, an automated fuzzer randomized combinations of extension headers and options and sent them to the victims. Then, after reviewing the responses to these fuzzed

packets, certain packets were manually crafted. Two parser differentials were discovered through this process that have not been documented in previous research.

Chained Extension Headers

The IPv6 standard does not state how many extension headers can be in a packet, so long as these extension headers are examined in series [21]. Because of this implementation, a kernel may be unaware of how many extension headers must be processed and could misallocate memory. This attack chains one-hundred destination options extension headers together between an IPv6 header and an ICMPv6 Echo Request, as shown in Figure 2.

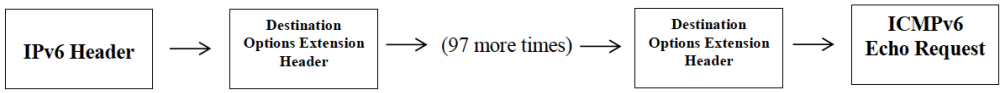

Figure 2: Packet Diagram of Chained Extension Headers

The bytes of this test packet are shown in Figure 3, where one of the Destination Options Extension Header is framed.

```
ff ff ff ff ff ff 08 00   27 09 32 3f 86 dd 60 00
00 00 03 28 3c 40 20 01   0d b8 00 00 f1 01 00 00
00 00 00 00 00 01 20 01   0d b8 00 00 f1 01 00 00
00 00 00 00 00 02 3c 00   01 04 00 00 00 00 3c 00
01 04 00 00 00 00 3c 00   01 04 00 00 00 00 3c 00
01 04 00 00 00 00 3c 00   01 04 00 00 00 00 3c 00
01 04 00 00 00 00 3c 00   01 04 00 00 00 00 3c 00
01 04 00 00 00 00 3c 00   01 04 00 00 00 00 3c 00
01 04 00 00 00 00 3c 00   01 04 00 00 00 00 3c 00
01 04 00 00 00 00 3c 00   01 04 00 00 00 00 3c 00
01 04 00 00 00 00 3c 00   01 04 00 00 00 00 3c 00
01 04 00 00 00 00 3c 00   01 04 00 00 00 00 3c 00
01 04 00 00 00 00 3c 00   01 04 00 00 00 00 3c 00
01 04 00 00 00 00 3c 00   01 04 00 00 00 00 3c 00
01 04 00 00 00 00 3c 00   01 04 00 00 00 00 3c 00
01 04 00 00 00 00 3c 00   01 04 00 00 00 00 3c 00
01 04 00 00 00 00 3c 00   01 04 00 00 00 00 3c 00
01 04 00 00 00 00 3c 00   01 04 00 00 00 00 3c 00
01 04 00 00 00 00 3c 00   01 04 00 00 00 00 3c 00
01 04 00 00 00 00 3c 00   01 04 00 00 00 00 3c 00
01 04 00 00 00 00 3c 00   01 04 00 00 00 00 3c 00
01 04 00 00 00 00 3c 00   01 04 00 00 00 00 3c 00
01 04 00 00 00 00 3c 00   01 04 00 00 00 00 3c 00
01 04 00 00 00 00 3c 00   01 04 00 00 00 00 3c 00
01 04 00 00 00 00 3c 00   01 04 00 00 00 00 3c 00
01 04 00 00 00 00 3c 00   01 04 00 00 00 00 3c 00
01 04 00 00 00 00 3c 00   01 04 00 00 00 00 3c 00
01 04 00 00 00 00 3c 00   01 04 00 00 00 00 3c 00
01 04 00 00 00 00 3c 00   01 04 00 00 00 00 3c 00
01 04 00 00 00 00 3c 00   01 04 00 00 00 00 3c 00
01 04 00 00 00 00 3c 00   01 04 00 00 00 00 3c 00
01 04 00 00 00 00 3c 00   01 04 00 00 00 00 3c 00
01 04 00 00 00 00 3c 00   01 04 00 00 00 00 3c 00
01 04 00 00 00 00 3c 00   01 04 00 00 00 00 3c 00
01 04 00 00 00 00 3c 00   01 04 00 00 00 00 3c 00
01 04 00 00 00 00 3c 00   01 04 00 00 00 00 3c 00
01 04 00 00 00 00 3c 00   01 04 00 00 00 00 3c 00
01 04 00 00 00 00 3c 00   01 04 00 00 00 00 3c 00
01 04 00 00 00 00 3c 00   01 04 00 00 00 00 3c 00
01 04 00 00 00 00 3c 00   01 04 00 00 00 00 3c 00
01 04 00 00 00 00 3c 00   01 04 00 00 00 00 3c 00
01 04 00 00 00 00 3c 00   01 04 00 00 00 00 3c 00
01 04 00 00 00 00 3c 00   01 04 00 00 00 00 3c 00
01 04 00 00 00 00 3c 00   01 04 00 00 00 00 3c 00
01 04 00 00 00 00 3c 00   01 04 00 00 00 00 3c 00
01 04 00 00 00 00 3c 00   01 04 00 00 00 00 3c 00
01 04 00 00 00 00 3c 00   01 04 00 00 00 00 3c 00
01 04 00 00 00 00 3c 00   01 04 00 00 00 00 3c 00
01 04 00 00 00 00 3c 00   01 04 00 00 00 00 3c 00
01 04 00 00 00 00 3c 00   01 04 00 00 00 00 3c 00
01 04 00 00 00 00 3c 00   01 04 00 00 00 00 3c 00
01 04 00 00 00 00 3c 00   01 04 00 00 00 00 3c 00
01 04 00 00 00 00 3c 00   01 04 00 00 00 00 3a 00
01 04 00 00 00 00 80 00   42 44 00 00 00 00
```

Figure 3: Raw Bytes of Chained Extension Headers

Bad Extension Header Option Length

Destination Options and Hop-by-Hop Extension Headers have an option field that can include a number of different options. One of these options, PadN, sets "two or more octets of padding into the options area of a header" [1]. This attack sets this option length field to 255 octets, but doesn't include any option data. Therefore, this test examines whether the kernel can successfully parse options within the extension headers. The packet diagram in Figure 4 depicts this structure, while Figure 5 shows the bytes of the packet itself, with the option length field framed. Figure 6 shows the format of a IPv6 Extension Header Option, as defined by RFC2460 [1].

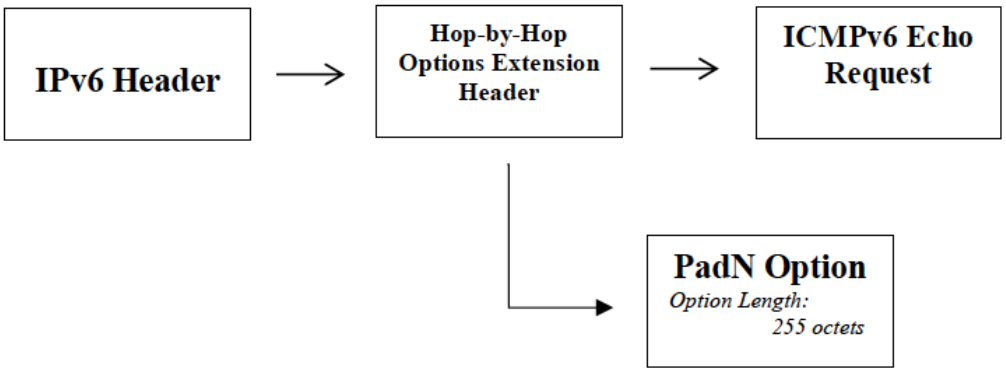

Figure 4: Packet Diagram of Bad Option Length

```
ff ff ff ff ff ff 08 00    27 09 32 3f 86 dd 60 00
00 00 00 10 00 40 20 01    0d b8 00 00 f1 01 00 00
00 00 00 00 00 01 20 01    0d b8 00 00 f1 01 00 00
00 00 00 00 00 02 3a 00    01 ff 01 02 00 00 80 00
42 44 00 00 00 00
```

Figure 5: Raw Bytes of Bad Option Length

```
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+- - - - - - - - -
|  Option Type  |  Opt Data Len |  Option Data
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+- - - - - - - - -

Option Type          8-bit identifier of the type of option.

Opt Data Len         8-bit unsigned integer.  Length of the Option
                     Data field of this option, in octets.

Option Data          Variable-length field.  Option-Type-specific
                     data.
```

Figure 6: TLV options (RFC 2460)

5 Results

Table 1 provides an overview of all of the results discovered in this paper.

Table 1: Results

Attack	FreeBSD	Linux
Systemic Vulnerabilities		
1. Neighbor Solicitation	*vulnerable*	*vulnerable*
2. Neighbor Discovery Protocol	*vulnerable*	*vulnerable*
3. Router Advertisement	*vulnerable*	*vulnerable*
4. MTU Poisoning	*not vulnerable*	*not vulnerable*
5. Destination Unreachable	*not vulnerable*	*not vulnerable*
6. Redirection	*not vulnerable*	*not vulnerable*
7. Ping of Death	*not vulnerable*	*not vulnerable*
8. New Rose	*not vulnerable*	*not vulnerable*
9. Land Attack	*not vulnerable*	*not vulnerable*
10. Source Smurfing	*not vulnerable*	*not vulnerable*
Parser Differentials		
11. Chained Extension Headers	*not vulnerable*	*vulnerable*
12. Extension Headers Options	*vulnerable*	*not vulnerable*

Each row in the table refers to one of the test cases that were tried in this experiment. Tests 1-10 were performed in previous papers on earlier versions of the kernels. Despite previous research's revelation of these systemic vulnerabilities, these results show that vulnerabilities still exist in Neighbor Solicitation, Neighbor Discovery Protocol, and Router Advertisements in both Ubuntu Linux 4.13 and FreeBSD 11.1. These vulnerabilities cannot be traced back to specific points in the kernel code, but are rather inherent in some configurations for the kernel. For instance, a node should update its routing table when it receives a router advertisement, so that it can find the fastest way for its packets to reach their destinations. When this feature is enabled, attackers can exploit it because there is currently no way to verify whether a Router Advertisement is forged, short of sending traffic with the router and hoping for a reply. With the current code, the only way to avoid this vulnerability is to configure the kernel to ignore Router Advertisements.

Tests 11 and 12 were a unique contribution of this paper that differed from the aforementioned tests. Specifically, they expose two parser differentials in the handling of IPv6 extension headers and options that are due to certain lines in the kernel source code. Using *dtrace* and *kprobe* allowed us to isolate where these vulnerabilities likely occur.

5.1 Chained Extension Headers

FreeBSD

FreeBSD does not respond to this malformed packet with the chained Destination Options Extension Headers, as shown in Figure 7. Other types of extension headers were also tried, but they yielded the same result. *Dtrace* was run with *ip6_input()* set as the breakpoint. Output, available in Appendix A.1, shows that this packet didn't make it into the network stack very far, before being discarded by the kernel.

```
root@:~/THESIS # tcpdump
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on em0, link-type EN10MB (Ethernet), capture size 262144 bytes
16:16:06.140274 IP6 2001:db8:0:f101::1 > 2001:db8:0:f101::2: DSTOPT DSTOPT DSTOP
T DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT D
STOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTO
PT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT
DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DST
OPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT
 DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DS
TOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOP
T DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT D
STOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT ICMP6, echo request, seq 0, length 8
^C
1 packet captured
1 packet received by filter
0 packets dropped by kernel
root@:~/THESIS #
```

Figure 7: FreeBSD *tcpdump* output, in response to chained extension headers

Linux

Linux is vulnerable to processing this malformed packet. When the Linux kernel receives an ICMPv6 Echo Request with one-hundred Destination Options Extension Headers, it responds with an ICMPv6 Echo Reply, shown in Figure 8. It also responded to different types of chained extension headers

with the same number in series, with the exception of the Hop-by-Hop Extension Header. By RFC 7045, there can only be one Hop-by-Hop Extension header in a packet and it must come immediately after the main IPv6 header [21].

```
root@osboxes:~/THESIS# tcpdump
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on enp0s3, link-type EN10MB (Ethernet), capture size 262144 bytes
16:53:04.897872 IP6 osboxes > fe80::a00:27ff:fe09:323f: ICMP6, neighbor solicitation, who has fe80::a00:2
7ff:fe09:323f, length 32
16:53:04.899064 IP6 fe80::a00:27ff:fe09:323f > osboxes: ICMP6, neighbor advertisement, tgt is fe80::a00:2
7ff:fe09:323f, length 24
16:53:05.023573 IP6 osboxes.42172 > fe80::a00:27ff:fe09:323f.hostmon: Flags [S], seq 3630443014, win 2880
0, options [mss 1440,sackOK,TS val 3872699007 ecr 0,nop,wscale 7], length 0
16:53:05.023784 IP6 fe80::a00:27ff:fe09:323f.hostmon > osboxes.42172: Flags [R.], seq 0, ack 3630443015,
win 0, length 0
16:53:05.710753 IP6 2001:db8:0:f101::1 > osboxes: DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT
 DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT
 DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT
 DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT
 DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT
 DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT
 DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT DSTOPT
 DSTOPT DSTOPT ICMP6, echo request, seq 0, length 8
16:53:05.710802 IP6 osboxes > 2001:db8:0:f101::1: ICMP6, echo reply, seq 0, length 8
16:53:05.832792 IP6 osboxes.59830 > 2001:db8:0:f101::1.hostmon: Flags [S], seq 2675961366, win 28800, opt
ions [mss 1440,sackOK,TS val 2311489842 ecr 0,nop,wscale 7], length 0
16:53:05.834834 IP6 2001:db8:0:f101::1.hostmon > osboxes.59830: Flags [R.], seq 0, ack 2675961367, win 0,
 length 0
^C
8 packets captured
8 packets received by filter
0 packets dropped by kernel
```

Figure 8: Linux *tcpdump* output, in response to chained extension headers

This is a clear instance of a parser differential between the FreeBSD and Linux kernels. The Linux kernel doesn't seem to limit the number of headers it examines in *ip6_input()*. Examining the output from *kprobe* in Appendix A.2 shows that the packet makes it through all the system calls without error. Specifically, *ip6_input_finish()* calls on *ipv6_destopt_rcv()* one-hundred times and then finishes by responding to the ICMPv6 Echo Request.

Looking at the kernel function, *ip6_input_finish()*, in Appendix A.3, one can see that the Linux kernel passes off the headers to a handler without considering how many headers are in the packet. The case *resubmit_final* in line 22 seems to get hit in every iteration and the handler in line 61 of the function makes the call to *ipv6_destopt_rcv()* that parses the IPv6 Destination Options Extension Header.

5.2 Bad Extension Header Option Length

FreeBSD

The FreeBSD kernel does respond to this packet with an ICMPv6 Echo Reply, as shown in Figure 9. Therefore, FreeBSD does not correctly verify the validity of these options. Appendix A.4 shows the output of *dtrace*, which confirms that the packet makes it through the network stack.

```
root@:~/THESIS # tcpdump
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on em0, link-type EN10MB (Ethernet), capture size 262144 bytes
21:20:13.809663 IP6 2001:db8:0:f101::1 > 2001:db8:0:f101::2: HBH ICMP6, echo req
uest, seq 0, length 8
21:20:13.809702 IP6 2001:db8:0:f101::2 > 2001:db8:0:f101::1: ICMP6, echo reply,
seq 0, length 8
^C
2 packets captured
2 packets received by filter
0 packets dropped by kernel
root@:~/THESIS #
```

Figure 9: FreeBSD *tcpdump* output, in response to bad option length

The call to process the option within the IPv6 Hop-by-Hop Extension Header must occur in the call *ip6_process_hopopts()*. In the code listed in Appendix A.5, line 29 shows that the code only verifies that the packet is not too small to be a valid option. It does not verify that the packet is the right size because it doesn't consider an upper bound.

Linux

Linux does not respond to this malformed packet, as shown in Figure 10. During the testing process, *kprobe* didn't even trigger with a breakpoint at *ip6_input()*, meaning that the packet was discarded upstream. *Kprobe* was reconfigured to trigger at *ip6_recv()*. The output from *kprobe*, available in Appendix A.6, confirms that this packet was discarded, somewhere in *ip6_parse_hopopts()*.

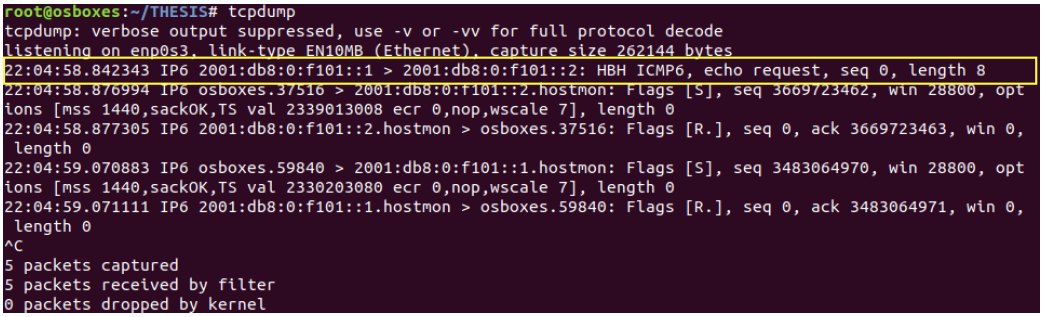

Figure 10: Linux *tcpdump* output, in response to bad option length

Code from *ipv6_rcv()*, available in Appendix A.7, shows where the call is made to *ipv6_parse_hopopts()*. If an error occurred in this function, which most likely happened, *ipv6_recv()* would drop the packet.

Appendix A.8 contains the code for *ipv6_parse_hopopts()*. In this code, it is clear to see that the Linux kernel does check the length of the options packet against the actual size of the option. Specifically, Line 22 of the code in Appendix A.8 shows that the function makes a call to *ip6_parse_tlv()*. Appendix A.9 shows that this function does properly check the length of a PadN option in lines 30-47. Because the malformed packet had an option length of 255 octets, line 37 identifies the packet as bad, such that it is discarded in *ipv6_recv()*.

6 Future Work

While this paper demonstrated that previously known vulnerabilities persist and introduced two new parser differentials, there is significant work yet to be done. There are two areas where this work could be built upon. First, one could attempt to patch the vulnerabilities that were discussed in the previous sections. However, some of the vulnerabilities, such as the systemic ones, may not have an easy fix because they are borne from logical missteps and not from processing errors. In these instances, the solution may be to make specific IPv6 features configurable. Turning off select features, however, is not optimal. More research must be done on how to address these systemic vulnerabilities.

On the other hand, the parser differentials described in sections **5.11** and

5.12 have a more direct solution. In the case of chained extension headers, destination options must only appear twice in a sequence of extension headers. Therefore, while Ubuntu Linux 4.13 mis-implemented the standard, it could be easily patched by inserting a line of code that tracks the number of extension headers it receives. A line of code could also effectively patch FreeBSD 11.1, in the case of the bad option length. In addition to examining how to handle the systemic vulnerabilities, future work should patch the source code and publish a more stable release.

The second way in which this research can be furthered is to develop an automatic fuzzing framework. Ideally, this framework would be intelligent in how it generates test cases and efficient in testing only meaningful cases. This is a challenging task because it is essential for the fuzzer to be run from a single machine and there is no easy way to integrate a feedback loop on one machine. This feedback loop would need to monitor the effectiveness of a malformed packet at penetrating the victim's stack on one machine, which is provided by current tracers. With current technology, however, a fuzzer can't access the output of a tracer, like *dtrace* and *kprobe*, which is on the victim's machine. If future research developed a new generation of fuzzers, it would facilitate and improve the testing of new kernels.

7 Conclusion

This paper built on the existing literature by re-evaluating known vulnerabilities from ten years ago, as well as offering up the discovery of two, new parser differentials. Despite the multitude of different releases of the FreeBSD kernels and Linux kernels since IPv6 was released, there are a number of alarming vulnerabilities in FreeBSD 11.1 and Ubuntu Linux 4.13 that could be used in damaging exploits. For instance, if a network intrusion detection (NID) marks a packet as invalid because it has a bad option length value, but the FreeBSD kernel accepts it, this packet could be used to insert malicious code that evades detection [22]. IPv6 is not fully tested, so there must be a number of parser differentials that haven't yet been discovered and could be exploited in the future. With the current tools and testing frameworks, it is difficult for those that patch the kernels to keep up with the pace of discovery of new vulnerabilities. As the amount of IPv6 traffic rises, it is paramount that researchers both patch existing vulnerabilities and develop improved testing frameworks in order to ensure the security and stability of

IPv6 moving forward.

8 Acknowledgements

This paper reflects the work of many individuals at Dartmouth. I would like to especially thank my advisor, Professor Sergey Bratus, for his wisdom and guidance throughout the past year. His course, Computer Networks, first sparked my interest in network security last spring. I am incredibly grateful for his willingness to help me explore the field.

I would also like to thank Michael Millian and Prashant Anantharaman for helping me revise this paper. They provided valuable insight on how to best present these findings. Additionally, Michael helped me to develop a script that scraped the CVE database for vulnerability reports. Finally, I would like to thank Professor Devin Balkcom, Professor Sean Smith, and Professor Xia Zhou for serving on my defense committee and for their feedback on this work.

References

[1] S. Deering and R. Hinden, "Internet Protocol, Version 6 (IPv6) Specification,"RFC 2460 (Draft Standard), IETF, December 1998.

[2] G. Jean-Malbuisson, "State of IPv6 Deployment 2017.," Internet Society, 2017.

[3] C. Pignataro and F. Gont, "Formally Deprecating Some IPv4 Options," RFC6814 (Proposed Standard), IETF, November 2012.

[4] J. Ullrich, K. Krombholz, H. Hobel, A. Dabrowski, and E. Weippl, "IPv6 Security: Attacks and Countermeasures in a Nutshell," SBA Research, August 2014.

[5] W. George, C. Donley, C. Lijenstolpe, and L. Howard, "IPv6 Support Required for All IP-Capable Nodes," RFC6540 (Best Current Practice), IETF, April 2012.

[6] C. Lecigne and G. Neville-Neil, "Walking Through the FreeBSD IPv6 Stack," FreeBSD, August 2006.

[7] Cve.mitre.org. (2018). CVE-CVE-2008-2476. [online] Available at: https://cve.mitre.org/cgi-bin-cvename.cgi?name=CVE-2008-2476 [Accessed 15 Jan. 2018].

[8] Cve.mitre.org. (2018). CVE-CVE-2007-1532. [online] Available at: https://cve.mitre.org/cgi-bin-cvename.cgi?name=CVE-2007-1532 [Accessed 20 Jan. 2018].

[9] R. Braden, "Requirements for Internet Hosts - Communication Layers," RFC1122 (Internet Standard), IETF, October 1989.

[10] F. Gont, "ICMP attacks against TCP," (Internet Draft), IETF, October 2006.

[11] J. McCann, S. Deering, and J. Mogul, "Path MTU Discovery for IP version 6," RFC1981 (Draft Standard), IETF, August 1996.

[12] Cve.mitre.org. (2018). CVE-CVE-2008-3530. [online] Available at: https://cve.mitre.org/cgi-bin-cvename.cgi?name=CVE-2008-3530 [Accessed 20 Jan. 2018].

[13] S. Kent and K. Seo, "Security Architecture for the Internet Protocol," RFC4301 (Proposed Standard), IETF, December 2005.

[14] J. Jeong, S. Park, L. Beloeil, and S. Madanapalli, "IPv6 Router Advertisement Options for DNS Configuration," RFC8106 (Proposed Standard), IETF, March 2017.

[15] Cve.mitre.org. (2018). CVE-CVE-2011-2393. [online] Available at: https://cve.mitre.org/cgi-bin-cvename.cgi?name=CVE-2011-2393 [Accessed 22 Jan. 2018].

[16] J. McCann, S. Deering, J. Mogul, and R. Hinden, "Path MTU discovery for IP version 6," RFC8201 (Internet Standard), IETF, July 2017.

[17] Cve.mitre.org. (2018). CVE-CVE-2015-8215. [online] Available at: https://cve.mitre.org/cgi-bin-cvename.cgi?name=CVE-2015-8215 [Accessed 08 Jan. 2018].

[18] F. Gont, "ICMP Attacks Against TCP," RFC5927 (Informational), IETF, July 2010.

[19] Cve.mitre.org. (2018). CVE-CVE-2005-0068. [online] Available at: https://cve.mitre.org/cgi-bin-cvename.cgi?name=CVE-2005-0068 [Accessed 02 Jan. 2018].

[20] Cve.mitre.org. (2018). CVE-CVE-2014-2144. [online] Available at: https://cve.mitre.org/cgi-bin-cvename.cgi?name=CVE-2014-2144 [Accessed 08 Jan. 2018].

[21] B. Carpenter and S. Jiang, "Transmission and Processing of IPv6 Extension Headers," RFC7045 (Proposed Standard), IETF, December 2013.

[22] M. Handley, V. Paxon, and C. Kreibich, "Network Intrusion Detection: Evasion, Traffic Normalization, and End-To-End Protocol Semantics," USENIX, August 2001.

Appendix

A.1

dtrace output of FreeBSD response to a chained extension header

```
CPU FUNCTION
  0  -> ip6_input
  0    -> ipsec6_capability
  0    <- ipsec6_capability
  0    -> in6_clearscope
  0    <- in6_clearscope
  0    -> in6_clearscope
  0    <- in6_clearscope
  0    -> in6_setscope
  0    <- in6_setscope
  0    -> in6_setscope
  0    <- in6_setscope
  0    -> in6ifa_ifwithaddr
  0      -> _rm_rlock
  0      <- _rm_rlock
  0      -> _rm_runlock
  0      <- _rm_runlock
  0    <- in6ifa_ifwithaddr          *** AT THIS POINT THE KERNEL
  0    -> m_freem                  *** HAS MARKED THE PACKET AS INVALID
  0      -> mb_free_ext
  0        -> uma_zfree_arg
  0          -> mb_dtor_pack
  0            -> m_tag_delete_chain
  0            <- m_tag_delete_chain
  0            -> uma_zone_exhausted_nolock
  0            <- uma_zone_exhausted_nolock
  0          <- mb_dtor_pack
  0          -> critical_enter
  0          <- critical_enter
  0          -> critical_exit
  0          <- critical_exit
  0        <- uma_zfree_arg
  0      <- mb_free_ext
  0    <- m_freem
```

A.2

kprobe output of Linux response to a chained extension header

```
# tracer: function_graph
#
# CPU  DURATION                  FUNCTION CALLS
# |     |   |                     |   |   |   |
 0)               |  ip6_input() {
 0)               |    ip6_input_finish() {
 0)   0.218 us    |      raw6_local_deliver();
 0)               |      ipv6_destopt_rcv() {
 0)   0.166 us    |        ip6_parse_tlv();
 0)   0.718 us    |      }

---------------- 99 more times -----------------

*** KERNEL FINISHES PROCESSING DESTINATION OPTIONS EXTENSION HEADERS HERE
*** AND PASSES THE PACKET ALONG


 0)               |      raw6_local_deliver() {
 0)   0.153 us    |        _raw_read_lock();
 0)   0.302 us    |        __raw_v6_lookup();
 0)               |        skb_clone() {
 0)   0.109 us    |          kmem_cache_alloc();
 0)               |          __skb_clone() {
 0)   0.160 us    |            __copy_skb_header();
 0)   0.638 us    |          }
 0)   1.673 us    |        }
 0)               |        rawv6_rcv() {
 0)   0.062 us    |          csum_ipv6_magic();
 0)               |          __skb_checksum_complete() {
 0)               |            skb_checksum() {
 0)               |              __skb_checksum() {
 0)               |                csum_partial() {
 0)   0.128 us    |                  do_csum();
 0)   0.623 us    |                }
 0)   1.230 us    |              }
 0)   1.687 us    |            }
 0)   2.363 us    |          }
 0)   0.053 us    |          dst_release();
 0)               |          sock_queue_rcv_skb() {
 0)               |            sk_filter_trim_cap() {
 0)               |              security_sock_rcv_skb() {
 0)   0.056 us    |                apparmor_socket_sock_rcv_skb();
```

```
0)   0.850 us    |              }
0)   1.410 us    |            }
0)               |            __sock_queue_rcv_skb() {
0)   0.053 us    |              _raw_spin_lock_irqsave();
0)   0.110 us    |              _raw_spin_unlock_irqrestore();
0)               |              sock_def_readable() {
0)               |                __wake_up_sync_key() {
0)   0.061 us    |                  _raw_spin_lock_irqsave();
0)               |                  __wake_up_common() {
0)               |                    pollwake() {
0)               |                      default_wake_function() {
0)               |                        try_to_wake_up() {
0)   0.382 us    |                          _raw_spin_lock_irqsave();
0)   0.111 us    |                          _raw_spin_lock();
0)   0.215 us    |                          update_rq_clock();
0)               |                          ttwu_do_activate() {
0)               |                            activate_task() {
0)   5.129 us    |                              enqueue_task_fair();
0)   5.849 us    |                            }
0)               |                            ttwu_do_wakeup() {
0)   0.357 us    |                              check_preempt_curr();
0)   2.040 us    |                            }
0)   8.785 us    |                          }
0)   0.070 us    |                          _raw_spin_unlock_irqrestore();
0) + 13.120 us   |                        }
0) + 13.515 us   |                      }
0) + 14.577 us   |                    }
0) + 15.332 us   |                  }
0)   0.060 us    |                  _raw_spin_unlock_irqrestore();
0) + 16.368 us   |                }
0) + 17.173 us   |              }
0) + 18.722 us   |            }
0) + 20.962 us   |          }
0) + 25.658 us   |        }
0)   0.054 us    |        __raw_v6_lookup();
0) + 30.018 us   |      }
0)               |      icmpv6_rcv() {
0)   0.042 us    |        csum_ipv6_magic();
0)               |        __skb_checksum_complete() {
0)               |          skb_checksum() {
0)               |            __skb_checksum() {
0)               |              csum_partial() {
0)   0.043 us    |                do_csum();
0)   0.300 us    |              }
0)   0.564 us    |            }
```

```
0)   0.880 us    |          }
0)   1.138 us    |        }
0)               |        icmpv6_echo_reply() {
0)   0.196 us    |          make_kuid();
0)   0.115 us    |          security_skb_classify_flow();
0)   0.326 us    |          _raw_spin_trylock();
0)               |          ip6_dst_lookup() {
0)               |            ip6_dst_lookup_tail() {
0)               |              ip6_route_output_flags() {
0)   0.040 us    |                __ipv6_addr_type();
0)   0.045 us    |                __ipv6_addr_type();
0)               |                fib6_rule_lookup() {
0)               |                  l3mdev_update_flow() {
0)   0.055 us    |                    dev_get_by_index_rcu();
0)   0.041 us    |                    l3mdev_master_ifindex_rcu();
0)   0.044 us    |                    dev_get_by_index_rcu();
0)   0.042 us    |                    l3mdev_master_ifindex_rcu();
0)   1.130 us    |                  }
0)               |                  fib_rules_lookup() {
0)   0.050 us    |                    fib6_rule_match();
0)               |                    fib6_rule_action() {
0)   0.071 us    |                      fib6_get_table();
0)               |                      ip6_pol_route_output() {
0)               |                        ip6_pol_route() {
0)   0.054 us    |                          _raw_read_lock_bh();
0)               |                          fib6_lookup() {
0)   0.157 us    |                            fib6_lookup_1();
0)   0.421 us    |                          }
0)               |                          find_match() {
0)   0.102 us    |                            rt6_check_expired();
0)   0.419 us    |                            rt6_score_route();
0)   1.070 us    |                          }
0)   0.080 us    |                          fib6_backtrack();
0)               |                          find_match() {
0)   0.040 us    |                            rt6_check_expired();
0)   0.044 us    |                            rt6_score_route();
0)   0.545 us    |                          }
0)   0.040 us    |                          fib6_backtrack();
0)               |                          _raw_read_unlock_bh() {
0)   0.041 us    |                            __local_bh_enable_ip();
0)   0.285 us    |                          }
0)   4.260 us    |                        }
0)   4.526 us    |                      }
0)   0.042 us    |                      dst_release();
0)   5.344 us    |                    }
```

```
0)   0.119 us    |                    fib6_rule_match();
0)               |                    fib6_rule_action() {
0)   0.106 us    |                      fib6_get_table();
0)               |                      ip6_pol_route_output() {
0)               |                        ip6_pol_route() {
0)   0.041 us    |                          _raw_read_lock_bh();
0)               |                          fib6_lookup() {
0)   0.348 us    |                            fib6_lookup_1();
0)   0.615 us    |                          }
0)               |                          find_match() {
0)   0.041 us    |                            rt6_check_expired();
0)   0.054 us    |                            rt6_score_route();
0)   0.687 us    |                          }
0)               |                          _raw_read_unlock_bh() {
0)   0.042 us    |                            __local_bh_enable_ip();
0)   0.287 us    |                          }
0)   2.613 us    |                        }
0)   2.868 us    |                      }
0)   3.447 us    |                    }
0)   0.095 us    |                    fib6_rule_suppress();
0) + 10.265 us   |                  }
0) + 11.882 us   |                }
0) + 12.660 us   |              }
0) + 12.960 us   |            }
0) + 13.287 us   |          }
0)   0.198 us    |          xfrm_lookup();
0)   0.164 us    |          ip6_dst_hoplimit();
0)               |          ip6_append_data() {
0)               |            ip6_setup_cork() {
0)   0.127 us    |              ip6_mtu();
0)   0.595 us    |            }
0)               |            __ip6_append_data.isra.39() {
0)               |              sock_alloc_send_skb() {
0)               |                sock_alloc_send_pskb() {
0)               |                  alloc_skb_with_frags() {
0)               |                    __alloc_skb() {
0)   0.140 us    |                      kmem_cache_alloc_node();
0)               |                      __kmalloc_reserve.isra.40() {
0)               |                        __kmalloc_node_track_caller() {
0)   0.171 us    |                          kmalloc_slab();
0)   0.889 us    |                        }
0)   1.233 us    |                      }
0)   0.495 us    |                      ksize();
0)   2.868 us    |                    }
0)   3.264 us    |                  }
```

```
0)   0.060 us    |                  skb_set_owner_w();
0)   4.037 us    |                }
0)   4.509 us    |              }
0)   0.042 us    |              skb_put();
0)   5.630 us    |            }
0)   6.853 us    |          }
0)               |          icmpv6_push_pending_frames() {
0)               |            csum_partial() {
0)   0.076 us    |              do_csum();
0)   0.324 us    |            }
0)   0.041 us    |            csum_ipv6_magic();
0)               |            ip6_push_pending_frames() {
0)               |              __ip6_make_skb() {
0)   0.041 us    |                skb_push();
0)               |                __get_hash_from_flowi6() {
0)   0.340 us    |                  flow_hash_from_keys();
0)   0.661 us    |                }
0)               |                ip6_cork_release.isra.38() {
0)   0.045 us    |                  dst_release();
0)   0.318 us    |                }
0)   2.330 us    |              }
0)               |              ip6_send_skb() {
0)               |                ip6_local_out() {
0)   0.047 us    |                  __ip6_local_out();
0)               |                  ip6_output() {
0)               |                    ip6_finish_output() {
0)   0.046 us    |                      ip6_mtu();
0)               |                      ip6_finish_output2() {
0)               |                        __neigh_create() {
0)               |                          __kmalloc() {
0)   0.045 us    |                            kmalloc_slab();
0)   0.449 us    |                          }
0)   0.045 us    |                          init_timer_key();
0)   0.395 us    |                          ndisc_constructor();
0)   0.045 us    |                          _raw_write_lock_bh();
0)   0.040 us    |                          ndisc_hash();
0)               |                          _raw_write_unlock_bh() {
0)   0.040 us    |                            __local_bh_enable_ip();
0)   0.293 us    |                          }
0)   3.680 us    |                        }
0)               |                        neigh_resolve_output() {
0)               |                          __neigh_event_send() {
0)   0.040 us    |                            _raw_write_lock_bh();
0)               |                            neigh_add_timer() {
0)   1.148 us    |                              mod_timer();
```

```
0)   1.500 us    |                            }
0)               |                            neigh_probe() {
0)   0.224 us    |                              skb_clone();
0) + 28.565 us   |                              ndisc_solicit();
0)   1.003 us    |                              kfree_skb();
0) + 31.052 us   |                            }
0)   0.067 us    |                            __local_bh_enable_ip();
0) + 33.956 us   |                          }
0) + 34.344 us   |                        }
0)   0.052 us    |                        __local_bh_enable_ip();
0) + 39.855 us   |                      }
0) + 40.494 us   |                    }
0) + 40.869 us   |                  }
0) + 41.542 us   |                }
0) + 41.875 us   |              }
0) + 44.796 us   |            }
0) + 46.048 us   |          }
0)   0.054 us    |          dst_release();
0)   0.051 us    |          __local_bh_enable_ip();
0) + 71.245 us   |        }
0)               |        kfree_skb() {
0)               |          skb_release_all() {
0)               |            skb_release_head_state() {
0)   0.049 us    |              dst_release();
0)   0.529 us    |            }
0)   0.071 us    |            skb_release_data();
0)   1.350 us    |          }
0)               |          kfree_skbmem() {
0)   0.092 us    |            kmem_cache_free();
0)   0.489 us    |          }
0)   2.598 us    |        }
0) + 76.989 us   |      }
0) ! 225.749 us  |    }
0) ! 226.466 us  |  }
------------------------------------------
0)    <idle>-0    =>  Network-580
------------------------------------------
```

A.3

ip6_input_finish() from Linux kernel code

```
1 static int ip6_input_finish(struct net *net, struct sock *sk,
    struct sk_buff *skb)
2 {
```

```
3   const struct inet6_protocol *ipprot;
4   struct inet6_dev *idev;
5   unsigned int nhoff;
6   int nexthdr;
7   bool raw;
8   bool have_final = false;
9
10  /*
11   *  Parse extension headers
12   */
13
14  rcu_read_lock();
15resubmit:
16  idev = ip6_dst_idev(skb_dst(skb));
17  if (!pskb_pull(skb, skb_transport_offset(skb)))
18    goto discard;
19  nhoff = IP6CB(skb)->nhoff;
20  nexthdr = skb_network_header(skb)[nhoff];
21
22resubmit_final: *** CODE SEEMS TO HIT THIS POINT EVERY TIME ***
23  raw = raw6_local_deliver(skb, nexthdr);
24  ipprot = rcu_dereference(inet6_protos[nexthdr]);
25  if (ipprot) {
26    int ret;
27
28    if (have_final) {
29      if (!(ipprot->flags & INET6_PROTO_FINAL)) {
30        /* Once we've seen a final protocol don't
31         * allow encapsulation on any non-final
32         * ones. This allows foo in UDP encapsulation
33         * to work.
34         */
35        goto discard;
36      }
37    } else if (ipprot->flags & INET6_PROTO_FINAL) {
38      const struct ipv6hdr *hdr;
39
40      /* Only do this once for first final protocol */
41      have_final = true;
42
43      /* Free reference early: we don't need it any more,
44         and it may hold ip_conntrack module loaded
45         indefinitely. */
46      nf_reset(skb);
47
```

```
48        skb_postpull_rcsum(skb, skb_network_header(skb),
49                           skb_network_header_len(skb));
50        hdr = ipv6_hdr(skb);
51        if (ipv6_addr_is_multicast(&hdr->daddr) &&
52            !ipv6_chk_mcast_addr(skb->dev, &hdr->daddr,
53            &hdr->saddr) &&
54            !ipv6_is_mld(skb, nexthdr, skb_network_header_len(skb)
    ))
55          goto discard;
56      }
57      if (!(ipprot->flags & INET6_PROTO_NOPOLICY) &&
58          !xfrm6_policy_check(NULL, XFRM_POLICY_IN, skb))
59        goto discard;
60      *** HANDLER THAT GETS HIT FOR EACH EXTENSION HEADER ***
61      ret = ipprot->handler(skb);
62      if (ret > 0) {
63        if (ipprot->flags & INET6_PROTO_FINAL) {
64          /* Not an extension header, most likely UDP
65           * encapsulation. Use return value as nexthdr
66           * protocol not nhoff (which presumably is
67           * not set by handler).
68           */
69          nexthdr = ret;
70          goto resubmit_final;
71        } else {
72          goto resubmit;
73        }
74      } else if (ret == 0) {
75        __IP6_INC_STATS(net, idev, IPSTATS_MIB_INDELIVERS);
76      }
77    } else {
78      if (!raw) {
79        if (xfrm6_policy_check(NULL, XFRM_POLICY_IN, skb)) {
80          __IP6_INC_STATS(net, idev,
81                          IPSTATS_MIB_INUNKNOWNPROTOS);
82          icmpv6_send(skb, ICMPV6_PARAMPROB,
83                      ICMPV6_UNK_NEXTHDR, nhoff);
84        }
85        kfree_skb(skb);
86      } else {
87        __IP6_INC_STATS(net, idev, IPSTATS_MIB_INDELIVERS);
88        consume_skb(skb);
89      }
90    }
91    rcu_read_unlock();
```

```
92    return 0;
93
94  discard:
95    __IP6_INC_STATS(net, idev, IPSTATS_MIB_INDISCARDS);
96    rcu_read_unlock();
97    kfree_skb(skb);
98    return 0;
99  }
```

A.4

dtrace output of FreeBSD response to a bad option length

```
CPU FUNCTION
  0  -> ip6_input
  0    -> ipsec6_capability
  0    <- ipsec6_capability
  0    -> in6_clearscope
  0    <- in6_clearscope
  0    -> in6_clearscope
  0    <- in6_clearscope
  0    -> in6_setscope
  0    <- in6_setscope
  0    -> in6_setscope
  0    <- in6_setscope
  0    -> in6ifa_ifwithaddr
  0      -> _rm_rlock
  0      <- _rm_rlock
  0      -> bcmp
  0      <- bcmp
  0      -> ifa_ref
  0      <- ifa_ref
  0      -> _rm_runlock
  0      <- _rm_runlock
  0    <- in6ifa_ifwithaddr
  0    -> ifa_free
  0    <- ifa_free
  0    -> ip6_process_hopopts
  0    <- ip6_process_hopopts
  0    -> ipsec6_input
  0      -> ipsec6_in_reject
  0        -> ipsec6_getpolicy
  0          -> ipsec_getpcbpolicy
  0          <- ipsec_getpcbpolicy
```

```
0          -> key_havesp
0          <- key_havesp
0          -> key_addref
0          <- key_addref
0        <- ipsec6_getpolicy
0        -> ipsec_in_reject
0        <- ipsec_in_reject
0        -> key_freesp
0        <- key_freesp
0      <- ipsec6_in_reject
0    <- ipsec6_input

*** AT THIS POINT THE KERNEL HAS PROCESSED THE ENTIRE IP HEADER ***

0    -> icmp6_input
0      -> in6_cksum
0        -> in6_cksum_partial
0          -> in6_getscope
0          <- in6_getscope
0          -> in6_getscope
0          <- in6_getscope
0        <- in6_cksum_partial
0        -> m_copym
0          -> uma_zalloc_arg
0            -> critical_enter
0            <- critical_enter
0            -> critical_exit
0            <- critical_exit
0            -> mb_ctor_mbuf
0              -> m_pkthdr_init
0                -> bzero
0                <- bzero
0                -> mac_mbuf_init
0                <- mac_mbuf_init
0              <- mb_ctor_mbuf
0            <- uma_zalloc_arg
0            -> m_tag_delete_chain
0            <- m_tag_delete_chain
0            -> m_tag_copy_chain
0            <- m_tag_copy_chain
0          <- m_copym
0          -> uma_zalloc_arg
0            -> critical_enter
0            <- critical_enter
0            -> critical_exit
```

```
0             <- critical_exit
0             -> mb_ctor_mbuf
0               -> m_pkthdr_init
0                 -> bzero
0                 <- bzero
0                 -> mac_mbuf_init
0                 <- mac_mbuf_init
0               <- mb_ctor_mbuf
0             <- uma_zalloc_arg
0             -> m_move_pkthdr
0               -> m_tag_delete_chain
0               <- m_tag_delete_chain
0             <- m_move_pkthdr
0             -> bcopy
0             <- bcopy
0             -> bcopy
0             <- bcopy
0             -> m_adj
0             <- m_adj
0             -> icmp6_reflect
0               -> in6ifa_ifwithaddr
0                 -> _rm_rlock
0                 <- _rm_rlock
0                 -> bcmp
0                 <- bcmp
0                 -> ifa_ref
0                 <- ifa_ref
0                 -> _rm_runlock
0                 <- _rm_runlock
0               <- in6ifa_ifwithaddr
0               -> ifa_free
0               <- ifa_free
0               -> in6_cksum
0                 -> in6_cksum_partial
0                   -> in6_getscope
0                   <- in6_getscope
0                   -> in6_getscope
0                   <- in6_getscope
0                 <- in6_cksum_partial
0                 -> ip6_output
0                   -> ipsec6_output
0                     -> ipsec6_common_output
0                       -> ipsec6_checkpolicy
0                         -> ipsec6_getpolicy
0                           -> ipsec_getpcbpolicy
```

```
0                         <- ipsec_getpcbpolicy
0                         -> key_havesp
0                         <- key_havesp
0                         -> key_addref
0                         <- key_addref
0                       <- ipsec6_getpolicy
0                       -> key_freesp
0                       <- key_freesp
0                     <- ipsec6_checkpolicy
0                   <- ipsec6_common_output
0                   -> bzero
0                   <- bzero
0                   -> bzero
0                   <- bzero
0                   -> bzero
0                   <- bzero
0                   -> in6_selectroute_fib
0                     -> selectroute
0                       -> bzero
0                       <- bzero
0                       -> in6_rtalloc1
0                         -> rtalloc1_fib
0                           -> __rw_rlock
0                           <- __rw_rlock
0                           -> rn_match
0                           <- rn_match
0                           -> _rw_runlock_cookie
0                           <- _rw_runlock_cookie
0                         <- rtalloc1_fib
0                       <- selectroute
0                     <- in6_selectroute_fib
0                     -> in6_setscope
0                     <- in6_setscope
0                     -> bzero
0                     <- bzero
0                     -> sa6_recoverscope
0                     <- sa6_recoverscope
0                     -> in6_setscope
0                     <- in6_setscope
0                     -> bzero
0                     <- bzero
0                     -> sa6_recoverscope
0                     <- sa6_recoverscope
0                     -> ip6_calcmtu
0                       -> bzero
```

```
0                       <- bzero
0                       -> tcp_hc_getmtu
0                         -> tcp_hc_lookup
0                         <- tcp_hc_lookup
0                       <- tcp_hc_getmtu
0                     <- ip6_calcmtu
0                     -> in6_clearscope
0                     <- in6_clearscope
0                     -> in6_clearscope
0                     <- in6_clearscope
0                     -> in6_ifawithifp
0                       -> in6_addrscope
0                       <- in6_addrscope
0                       -> __rw_rlock
0                       <- __rw_rlock
0                       -> in6_addrscope
0                       <- in6_addrscope
0                       -> in6_addrscope
0                       <- in6_addrscope
0                       -> ifa_ref
0                       <- ifa_ref
0                       -> _rw_runlock_cookie
0                       <- _rw_runlock_cookie
0                     <- in6_ifawithifp
0                     -> ifa_free
0                     <- ifa_free
0                     -> nd6_output_ifp
0                       -> mac_netinet6_nd6_send
0                       <- mac_netinet6_nd6_send
0                       -> ether_output
0                         -> mac_ifnet_check_transmit
0                         <- mac_ifnet_check_transmit
0                         -> nd6_resolve
0                           -> __rw_rlock
0                           <- __rw_rlock
0                           -> bzero
0                           <- bzero
0                           -> in6_lltable_lookup
0                             -> bcmp
0                             <- bcmp
0                           <- in6_lltable_lookup
0                           -> bcopy
0                           <- bcopy
0                           -> _rw_runlock_cookie
0                           <- _rw_runlock_cookie
```

```
0                       <- nd6_resolve
0                       -> m_prepend
0                         -> uma_zalloc_arg
0                           -> critical_enter
0                           <- critical_enter
0                           -> critical_exit
0                           <- critical_exit
0                           -> mb_ctor_mbuf
0                             -> m_pkthdr_init
0                               -> bzero
0                               <- bzero
0                               -> mac_mbuf_init
0                               <- mac_mbuf_init
0                             <- mb_ctor_mbuf
0                           <- uma_zalloc_arg
0                           -> m_tag_delete_chain
0                           <- m_tag_delete_chain
0                         <- m_prepend
0                         -> memcpy
0                         <- memcpy
0                         -> if_transmit
0                           -> lem_start
0                             -> if_getsoftc
0                             <- if_getsoftc
0                             -> if_getdrvflags
0                             <- if_getdrvflags
0                             -> lem_start_locked
0                               -> if_getsoftc
0                               <- if_getsoftc
0                               -> if_getdrvflags
0                               <- if_getdrvflags
0                               -> if_sendq_empty
0                               <- if_sendq_empty
0                               -> if_dequeue
0                               <- if_dequeue
0                               -> bus_dmamap_load_mbuf_sg
0                                 -> _bus_dmamap_load_buffer
0                                   -> bounce_bus_dmamap_load_buffer
0                                     -> pmap_kextract
0                                     <- pmap_kextract
0                                   <- bounce_bus_dmamap_load_buffer
0                                   -> _bus_dmamap_load_buffer
0                                     -> bounce_bus_dmamap_load_buffer
0                                       -> pmap_kextract
0                                       <- pmap_kextract
```

```
0                                       <- bounce_bus_dmamap_load_buffer
0                                       -> _bus_dmamap_complete
0                                         -> bounce_bus_dmamap_complete
0                                         <- bounce_bus_dmamap_complete
0                                       <- bus_dmamap_load_mbuf_sg
0                                       -> if_etherbpfmtap
0                                         -> bpf_mtap
0                                           -> m_length
0                                           <- m_length
0                                           -> __rw_rlock
0                                           <- __rw_rlock
0                                           -> bpf_filter
0                                             -> bzero
0                                             <- bzero
0                                           <- bpf_filter
0                                           -> _rw_runlock_cookie
0                                           <- _rw_runlock_cookie
0                                         <- bpf_mtap
0                                         -> if_sendq_empty
0                                         <- if_sendq_empty
0                                       <- lem_start_locked
0                                     <- lem_start
0                                   <- if_transmit
0                                 <- ether_output
0                                 -> rtfree
0                                 <- rtfree
0                               <- ip6_output
0                             <- icmp6_reflect
0                             -> bzero
0                             <- bzero
0                             -> sa6_recoverscope
0                             <- sa6_recoverscope
0                             -> __rw_rlock
0                             <- __rw_rlock
0                             -> __rw_rlock
0                             <- __rw_rlock
0                             -> _rw_runlock_cookie
0                             <- _rw_runlock_cookie
0                             -> _rw_runlock_cookie
0                             <- _rw_runlock_cookie
0                             -> m_freem
0                               -> mb_free_ext
0                               <- mb_free_ext
0                             <- m_freem
0                           <- icmp6_input
```

```
0                                   -> ip6_input
0                                     -> ipsec6_capability
0                                     <- ipsec6_capability
0                                     -> in6_clearscope
0                                     <- in6_clearscope
0                                     -> in6_clearscope
0                                     <- in6_clearscope
0                                     -> in6_setscope
0                                     <- in6_setscope
0                                     -> in6_setscope
0                                     <- in6_setscope
0                                     -> in6ifa_ifwithaddr
0                                       -> _rm_rlock
0                                       <- _rm_rlock
0                                       -> bcmp
0                                       <- bcmp
0                                       -> ifa_ref
0                                       <- ifa_ref
0                                       -> _rm_runlock
0                                       <- _rm_runlock
0                                     <- in6ifa_ifwithaddr
0                                     -> ifa_free
0                                     <- ifa_free
0                                     -> ipsec6_input
0                                       -> ipsec6_in_reject
0                                         -> ipsec6_getpolicy
0                                           -> ipsec_getpcbpolicy
0                                           <- ipsec_getpcbpolicy
0                                           -> key_havesp
0                                           <- key_havesp
0                                           -> key_addref
0                                           <- key_addref
0                                         <- ipsec6_getpolicy
0                                         -> ipsec_in_reject
0                                         <- ipsec_in_reject
0                                         -> key_freesp
0                                         <- key_freesp
0                                       <- ipsec6_in_reject
0                                     <- ipsec6_input
0                                     -> icmp6_input
0                                       -> in6_cksum
0                                         -> in6_cksum_partial
0                                           -> in6_getscope
0                                           <- in6_getscope
0                                           -> in6_getscope
```

```
0                                       <- in6_getscope
0                                     <- in6_cksum_partial
0                                     -> m_copym
0                                       -> uma_zalloc_arg
0                                         -> critical_enter
0                                         <- critical_enter
0                                         -> critical_exit
0                                         <- critical_exit
0                                         -> mb_ctor_mbuf
0                                           -> m_pkthdr_init
0                                             -> bzero
0                                             <- bzero
0                                             -> mac_mbuf_init
0                                             <- mac_mbuf_init
0                                           <- mb_ctor_mbuf
0                                         <- uma_zalloc_arg
0                                         -> m_tag_delete_chain
0                                         <- m_tag_delete_chain
0                                         -> m_tag_copy_chain
0                                         <- m_tag_copy_chain
0                                       <- m_copym
0                                       -> nd6_na_input
0                                         -> bzero
0                                         <- bzero
0                                         -> in6_setscope
0                                         <- in6_setscope
0                                         -> nd6_option_init
0                                           -> bzero
0                                           <- bzero
0                                         <- nd6_option_init
0                                         -> nd6_options
0                                         <- nd6_options
0                                         -> in6ifa_ifpwithaddr
0                                           -> __rw_rlock
0                                           <- __rw_rlock
0                                           -> bcmp
0                                           <- bcmp
0                                           -> bcmp
0                                           <- bcmp
0                                           -> _rw_runlock_cookie
0                                           <- _rw_runlock_cookie
0                                         <- in6ifa_ifpwithaddr
0                                         -> __rw_rlock
0                                         <- __rw_rlock
0                                         -> nd6_lookup
```

```
0                                   -> bzero
0                                   <- bzero
0                                   -> in6_lltable_lookup
0                                     -> bcmp
0                                     <- bcmp
0                                   <- in6_lltable_lookup
0                                 <- nd6_lookup
0                                 -> _rw_runlock_cookie
0                                 <- _rw_runlock_cookie
0                                 -> nd6_llinfo_setstate
0                                   -> nd6_llinfo_settimer_locked
0                                     -> callout_reset_sbt_on
0                                       -> callout_when
0                                       <- callout_when
0                                       -> callout_lock
0                                         -> spinlock_enter
0                                           -> critical_enter
0                                           <- critical_enter
0                                       <- callout_lock
0                                       -> callout_cc_add
0                                       <- callout_cc_add
0                                       -> spinlock_exit
0                                         -> critical_exit
0                                         <- critical_exit
0                                       <- spinlock_exit
0                                     <- callout_reset_sbt_on
0                                   <- nd6_llinfo_settimer_locked
0                                   <- nd6_llinfo_setstate
0                                   -> m_freem
0                                     -> mb_free_ext
0                                     <- mb_free_ext
0                                   <- m_freem
0                                 <- nd6_na_input
0                                 -> bzero
0                                 <- bzero
0                                 -> sa6_recoverscope
0                                 <- sa6_recoverscope
0                                 -> __rw_rlock
0                                 <- __rw_rlock
0                                 -> __rw_rlock
0                                 <- __rw_rlock
0                                 -> _rw_runlock_cookie
0                                 <- _rw_runlock_cookie
0                                 -> _rw_runlock_cookie
0                                 <- _rw_runlock_cookie
```

```
0                                       -> m_freem
0                                         -> mb_free_ext
0                                           -> uma_zfree_arg
0                                             -> mb_dtor_pack
0                                               -> m_tag_delete_chain
0                                               <- m_tag_delete_chain
0                                               -> uma_zone_exhausted_nolock
0                                               <- uma_zone_exhausted_nolock
0                                             <- mb_dtor_pack
0                                             -> critical_enter
0                                             <- critical_enter
0                                             -> critical_exit
0                                             <- critical_exit
0                                           <- uma_zfree_arg
0                                           -> uma_zfree_arg
0                                             -> mb_dtor_mbuf
0                                             <- mb_dtor_mbuf
0                                             -> critical_enter
0                                             <- critical_enter
0                                             -> critical_exit
0                                             <- critical_exit
0                                           <- uma_zfree_arg
0                                         <- m_freem
0                                       <- icmp6_input
```

A.5

ip6_process_hopopts() from FreeBSD kernel source code

```
1 /*
2  * Search header for all Hop-by-hop options and process each
     option.
3  * This function is separate from ip6_hopopts_input() in order
     to
4  * handle a case where the sending node itself process its hop-
     by-hop
5  * options header. In such a case, the function is called from
     ip6_output().
6  *
7  * The function assumes that hbh header is located right after
     the IPv6 header
8  * (RFC2460 p7), opthead is pointer into data content in m, and
     opthead to
```

```
 * opthead + hbhlen is located in contiguous memory region.
 */
int
ip6_process_hopopts(struct mbuf *m, u_int8_t *opthead, int
    hbhlen,
     u_int32_t *rtalertp, u_int32_t *plenp)
{
  struct ip6_hdr *ip6;
  int optlen = 0;
  u_int8_t *opt = opthead;
  u_int16_t rtalert_val;
  u_int32_t jumboplen;
  const int erroff = sizeof(struct ip6_hdr) + sizeof(struct
    ip6_hbh);

  for (; hbhlen > 0; hbhlen -= optlen, opt += optlen) {
    switch (*opt) {
    case IP6OPT_PAD1:
      optlen = 1;
      break;
    case IP6OPT_PADN:
    *** OPTION LENGTH VERIFICATION HAPPENS HERE ***
      if (hbhlen < IP6OPT_MINLEN) {
        IP6STAT_INC(ip6s_toosmall);
        goto bad;
      }
      optlen = *(opt + 1) + 2;
      break;
    case IP6OPT_ROUTER_ALERT:
      /* XXX may need check for alignment */
      if (hbhlen < IP6OPT_RTALERT_LEN) {
        IP6STAT_INC(ip6s_toosmall);
        goto bad;
      }
      if (*(opt + 1) != IP6OPT_RTALERT_LEN - 2) {
        /* XXX stat */
        icmp6_error(m, ICMP6_PARAM_PROB,
            ICMP6_PARAMPROB_HEADER,
             erroff + opt + 1 - opthead);
        return (-1);
      }
      optlen = IP6OPT_RTALERT_LEN;
      bcopy((caddr_t)(opt + 2), (caddr_t)&rtalert_val, 2);
      *rtalertp = ntohs(rtalert_val);
      break;
```

```
52      case IP6OPT_JUMBO:
53        /* XXX may need check for alignment */
54        if (hbhlen < IP6OPT_JUMBO_LEN) {
55          IP6STAT_INC(ip6s_toosmall);
56          goto bad;
57        }
58        if (*(opt + 1) != IP6OPT_JUMBO_LEN - 2) {
59          /* XXX stat */
60          icmp6_error(m, ICMP6_PARAM_PROB,
61              ICMP6_PARAMPROB_HEADER,
62               erroff + opt + 1 - opthead);
63          return (-1);
64        }
65        optlen = IP6OPT_JUMBO_LEN;
66
67        /*
68         * IPv6 packets that have non 0 payload length
69         * must not contain a jumbo payload option.
70         */
71        ip6 = mtod(m, struct ip6_hdr *);
72        if (ip6->ip6_plen) {
73          IP6STAT_INC(ip6s_badoptions);
74          icmp6_error(m, ICMP6_PARAM_PROB,
75              ICMP6_PARAMPROB_HEADER,
76               erroff + opt - opthead);
77          return (-1);
78        }
79
80        /*
81         * We may see jumbolen in unaligned location, so
82         * we'd need to perform bcopy().
83         */
84        bcopy(opt + 2, &jumboplen, sizeof(jumboplen));
85        jumboplen = (u_int32_t)htonl(jumboplen);
86
87  #if 1
88        /*
89         * if there are multiple jumbo payload options,
90         * *plenp will be non-zero and the packet will be
91         * rejected.
92         * the behavior may need some debate in ipngwg -
93         * multiple options does not make sense, however,
94         * there's no explicit mention in specification.
95         */
96        if (*plenp != 0) {
```

```
97          IP6STAT_INC(ip6s_badoptions);
98          icmp6_error(m, ICMP6_PARAM_PROB,
99              ICMP6_PARAMPROB_HEADER,
100             erroff + opt + 2 - opthead);
101         return (-1);
102       }
103 #endif
104
105       /*
106        * jumbo payload length must be larger than 65535.
107        */
108       if (jumboplen <= IPV6_MAXPACKET) {
109         IP6STAT_INC(ip6s_badoptions);
110         icmp6_error(m, ICMP6_PARAM_PROB,
111             ICMP6_PARAMPROB_HEADER,
112             erroff + opt + 2 - opthead);
113         return (-1);
114       }
115       *plenp = jumboplen;
116
117       break;
118     default:    /* unknown option */
119       if (hbhlen < IP6OPT_MINLEN) {
120         IP6STAT_INC(ip6s_toosmall);
121         goto bad;
122       }
123       optlen = ip6_unknown_opt(opt, m,
124           erroff + opt - opthead);
125       if (optlen == -1)
126         return (-1);
127       optlen += 2;
128       break;
129     }
130   }
131
132   return (0);
133
134   bad:
135   m_freem(m);
136   return (-1);
137 }
```

A.6

kprobe output of Linux response to a bad option length

```
# tracer: function_graph
#
# CPU  DURATION                  FUNCTION CALLS
# |     |   |                     |   |   |   |
 0)               |  ipv6_rcv() {
 0)               |    ipv6_parse_hopopts() {

*** PROCESSES LENGTH HERE ***

 0)               |      ip6_parse_tlv() {
 0)               |        kfree_skb() {
 0)               |          skb_release_all() {
 0)   0.109 us    |            skb_release_head_state();
 0)               |            skb_release_data() {
 0)               |              skb_free_head() {
 0)   0.683 us    |                page_frag_free();
 0)   1.214 us    |              }
 0)   1.669 us    |            }
 0)   2.694 us    |          }
 0)               |          kfree_skbmem() {
 0)   0.203 us    |            kmem_cache_free();
 0)   0.716 us    |          }
 0)   4.180 us    |        }
 0)   4.677 us    |      }
 0)   5.303 us    |    }
 0)   7.340 us    |  }
```

A.7

Excerpt from ip_recv() from Linux kernel source code

```
1
2   \*
3   * from ipv6_recv() from ip6_input.c
4   *\
5   if (hdr->nexthdr == NEXTHDR_HOP) {
6     if (ipv6_parse_hopopts(skb) < 0) {   *** CHECK MADE HERE ***
7       __IP6_INC_STATS(net, idev, IPSTATS_MIB_INHDRERRORS);
8       rcu_read_unlock();
9       return NET_RX_DROP;
10    }
```

```
11   }
```

A.8

ipv6_parse_hopopts() from Linux kernel source code

```
1  int ipv6_parse_hopopts(struct sk_buff *skb)
2  {
3    struct inet6_skb_parm *opt = IP6CB(skb);
4
5    /*
6     * skb_network_header(skb) is equal to skb->data, and
7     * skb_network_header_len(skb) is always equal to
8     * sizeof(struct ipv6hdr) by definition of
9     * hop-by-hop options.
10    */
11   if (!pskb_may_pull(skb, sizeof(struct ipv6hdr) + 8) ||
12       !pskb_may_pull(skb, (sizeof(struct ipv6hdr) +
13           ((skb_transport_header(skb)[1] + 1) << 3)))) {
14     kfree_skb(skb);
15     return -1;
16   }
17
18   opt->flags |= IP6SKB_HOPBYHOP;
19
20    *** CHECKING FOR LENGTH HERE ***
21
22   if (ip6_parse_tlv(tlvprochopopt_lst, skb)) {
23     skb->transport_header += (skb_transport_header(skb)[1] + 1)
     << 3;
24     opt = IP6CB(skb);
25     opt->nhoff = sizeof(struct ipv6hdr);
26     return 1;
27   }
28   return -1;
29 }
```

A.9

ip6_parse_tlv() from Linux kernel source code

```
1
2  /* Parse tlv encoded option header (hop-by-hop or destination)
     */
```

```
3
4  static bool ip6_parse_tlv(const struct tlvtype_proc *procs,
       struct sk_buff *skb)
5  {
6    const struct tlvtype_proc *curr;
7    const unsigned char *nh = skb_network_header(skb);
8    int off = skb_network_header_len(skb);
9    int len = (skb_transport_header(skb)[1] + 1) << 3;
10   int padlen = 0;
11
12   if (skb_transport_offset(skb) + len > skb_headlen(skb))
13     goto bad;
14
15   off += 2;
16   len -= 2;
17
18   while (len > 0) {
19     int optlen = nh[off + 1] + 2;
20     int i;
21
22     switch (nh[off]) {
23     case IPV6_TLV_PAD1:
24       optlen = 1;
25       padlen++;
26       if (padlen > 7)
27         goto bad;
28       break;
29
30     case IPV6_TLV_PADN:    *** CHECKS LENGTHS HERE ***
31       /* RFC 2460 states that the purpose of PadN is
32        * to align the containing header to multiples
33        * of 8. 7 is therefore the highest valid value.
34        * See also RFC 4942, Section 2.1.9.5.
35        */
36       padlen += optlen;
37       if (padlen > 7)
38         goto bad;
39       /* RFC 4942 recommends receiving hosts to
40        * actively check PadN payload to contain
41        * only zeroes.
42        */
43       for (i = 2; i < optlen; i++) {
44         if (nh[off + i] != 0)
45           goto bad;
46       }
```

```
47        break;
48
49      default: /* Other TLV code so scan list */
50        if (optlen > len)
51          goto bad;
52        for (curr = procs; curr->type >= 0; curr++) {
53          if (curr->type == nh[off]) {
54            /* type specific length/alignment
55               checks will be performed in the
56               func(). */
57            if (curr->func(skb, off) == false)
58              return false;
59            break;
60          }
61        }
62        if (curr->type < 0) {
63          if (ip6_tlvopt_unknown(skb, off) == 0)
64            return false;
65        }
66        padlen = 0;
67        break;
68      }
69      off += optlen;
70      len -= optlen;
71    }
72
73    if (len == 0)
74      return true;
75  bad:
76    kfree_skb(skb);
77    return false;
78  }
```